```
//Hannah, Emma, Lindsey
//graph class

#include "vertex.h"
#include "list.h"
#include "pq.h"
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>


using namespace std;

class Graph
{
public:
          Graph         (string filename);
          Graph         (const Graph& g);
          ~Graph        (void);
  Graph   operator=     (const Graph& g);
  void    dfs           (void);
  bool    cycle         (void);
  void    Prim          (int root);


private:
  List<Vertex>  graph;
  List<Edge>    edges;
  bool          BALLS = false;

  void    dfs_visit       (Vertex &u, int timee);
  bool    pqHelp          (Vertex s, MinPriorityQueue<Vertex> pq);
  int     findEdge        (Vertex u, Vertex v);
};
//===========================================================================

//===========================================================================
//default constructor
//Pre-Condition:
//  -file with matrix representation of a graph
//Post-Condition:
//  -a graph
//===========================================================================
Graph::Graph  (string filename)
{
  ifstream file;
  file.open(filename);                                  //open file
  string line;
  getline(file, line);
  istringstream buffer(line);                           //read in number of
  int num_vert;                                         //vertices
  buffer >> num_vert;
  for (int i = 0; i < num_vert; i++)
  {
    Vertex *v = new Vertex(i);
    graph.append(v);                                    //append all vertices to graph
  }
  for (int j = 0; j < num_vert; j++)                    //iterates rows
  {
    int srch_pt = 0;
    getline(file, line);
    for (int k = 0; k <num_vert; k++)
```

```
    {
      int space = line.find(" ", srch_pt);              //read in connections
      int weight;
      istringstream buffer (line.substr(srch_pt, space-srch_pt));
      buffer >> weight;
      srch_pt = space+1;

      if (weight != 0)
      {
        Edge *e = new Edge();                            //creates edges
        e->u = graph[j]->id;
        e->v = graph[k]->id;
        e->weight = weight;
        edges.append(e);
        graph[j]->adj_list.append(graph[k]);
      }
    }
  }
  //cout << graph;
  file.close();
}

//==========================================================================
//copy constructor
//Pre-Conditions
//  -graph
//Post-Conditions
//  -copied graph
//==========================================================================
Graph::Graph  (const Graph& g)
{
  graph = g.graph;
}

//==========================================================================
//destructor
//Pre-Conditions
// - graph
//Post-Conditions
//  -no graph
//==========================================================================
Graph::~Graph (void)
{
  //delete this;
}

//==========================================================================
//assignment operator
//Pre-Conditions
//  -two different graphs
//Post-Conditions
//  -two same graphs
//==========================================================================
Graph Graph::operator= (const Graph& g)
{
  delete this;                //destructor
  graph = g.graph;            //copy
  edges = g.edges;
  return *this;
}

//==========================================================================
//dfs - depth first search
```

```
//Pre-Conditions
//  -graph
//Post-Conditions
//  -traversed graph
//===========================================================================
void Graph::dfs (void)
{
  if (graph.length()==0)                            //throw error if empty
    throw EmptyError();
  for (int i = 0; i < graph.length(); i++)
  {
    graph[i]->color = 'w';                          //set all to white
  }
  int timee = 0;
  //cout << timee << endl;
  for (int i = 0; i < graph.length(); i++)
  {
    if (graph[i]->color == 'w')
    {
      dfs_visit(*(graph[i]), timee);                //visit vertex
    }
  }
}
//===========================================================================
//dfs_visit - depth first search
//===========================================================================
void Graph::dfs_visit (Vertex& u, int timee)
{
  timee += 1;
  u.disc = timee;
  u.color = 'g';                                   //discover
  cout << "Visiting: " <<u << endl;
  for (int i = 0; i < u.adj_list.length(); i++)
  {
    if (u.adj_list[i]->color == 'g')
    {
      BALLS = true;
    }
    if (u.adj_list[i]->color == 'w')
    {
      u.adj_list[i]->pred = &u;                    //set predecessor
      dfs_visit(*u.adj_list[i], timee);            //visit adjacent vertices
    }
  }
  u.color = 'b';                                   //all adjacent vertices visited
  timee += 1;
  u.fin = timee;
}
//===========================================================================
//cycle
//Pre-Conditions
//  -graph
//Post-Conditions
//  -indication if graph contains cycle
//===========================================================================
bool Graph::cycle (void)
{
  dfs();
  return BALLS;

}

//===========================================================================
```

```
//Prim’s algorithm -- we tried but it’s not happening
//Pre-Conditions
//  -undirected weighted graph
//Post-Conditions
//  -MST of graph
//===========================================================================
void Graph::Prim (int root)
{
  MinPriorityQueue<Vertex> pq;                            //create min pq
  for (int i = 0; i < graph.length(); i++)
  {
    if (graph[i]->id != graph[root]->id)
    {
      graph[i]->key = INT_MAX;
      graph[i]->pred = NULL;
      pq.insert(graph[i]);                                //insert vertices into pq
    }
  }
  graph[root]->key = 0;
  pq.insert(graph[root]);                                 //insert root into pq
  cout << pq << endl;
  //cout << "SHIT" << endl;
  while (!pq.empty())
  {
    Vertex *u = pq.extractMin();                          //find minimum weight
    cout << "MIN: " << u->id << endl;
    for (int j = 0; j < u->adj_list.length(); j++)    //visit adjacent vertices
    {
      Vertex *v = u->adj_list[j];
      //cout << "weight of " << *v <<" " <<  v->key << endl;
      if (v->key > u->key && findEdge(u,v) < v->key)
      {
        //cout << "what to heck" << endl;
        v->pred = u;                                      //update pred and key
        v->key = findEdge(u,v);
        //cout << "reset key of " << v->id << "to " << v->key << endl;
      }
    }
  }
  for (int k = 0; k < graph.length(); k++)
  {
    cout << k << " weight: " << graph[k]->key <<graph[k]->pred <<endl;
  }
}

//===========================================================================
//pqHelp
// this method verifies if an element is present in the pq
//===========================================================================
bool Graph::pqHelp(Vertex s, MinPriorityQueue<Vertex> pq)
{
  while (!pq.empty())
  {
    Vertex v = pq.extractMin();
    if (v.id == s.id )
      return true;
  }
  return false;
}
//===========================================================================
//findEdge
// this method finds the edges and returns the weight given two vertices
//===========================================================================
```

```
int Graph::findEdge (Vertex u, Vertex v)
{
  for (int i = 0; i < edges.length(); i++)
  {
    if (edges[i]->u == u.id && edges[i]->v == v.id)
    {
      return edges[i]->weight;
    }
  }
  return 0;
}
```

```
//Hannah, Emma, Lindsey
//Vertex class

#include "list.h"
#include <climits>
using namespace std;

class Vertex
{
public:
  int id;
  Vertex* pred;
  List<Vertex> adj_list;
  char color;
  int disc;
  int fin;
  int key;

  bool    operator<     (const Vertex &x) {return this->key <= x.key;}
  bool    operator>     (const Vertex &x) {return this->key > x.key;}
  bool    operator==    (const Vertex &x) {return this->id == x.id;}

  Vertex& operator= (const Vertex& v);

          Vertex        (int name);
          Vertex        (Vertex* v);
          ~Vertex       (void);
private:
  friend ostream& operator<< (ostream& os, const Vertex& v)
  {
    os << "Vertex ID: " << v.id << endl;
    return os;
  }
};

Vertex::Vertex(int name)
{
  id = name;
  pred = NULL;
  color = 'w';
  disc = INT_MAX;
  fin = 0;
  key = INT_MAX;
}

Vertex::Vertex(Vertex* v)
{
  id = v->id;
  pred = v->pred;
  adj_list = v->adj_list;
  color = v->color;
  disc = v->disc;
  fin = v->fin;
}

Vertex::~Vertex ()
{
}

Vertex& Vertex::operator= (const Vertex& v)
{
  this->id = v.id;
  this->pred = v.pred;
```

```
  this->adj_list = v.adj_list;
  this->color = v.color;
  this->disc = v.disc;
  this->fin = v.fin;

  return *this;
}

class Edge
{
public:
  int u;    //start Vertex
  int v;    //end Vertex
  int weight;     //weight (u,v)

  Edge()
  {
    u = -1;
    v = -1;
    weight = 0;
  }

  ~Edge()
  {
  }
};
```

```
/*
* testin dis graph bish
*/



#include <iostream>
#include "graph.h"
using namespace std;



void test1 (void)
{
  cout << "testing grap.txt" << endl;
  Graph g1("grap.txt");
  //g1.dfs();
  cout << boolalpha << g1.cycle() << endl;
}


void test2 (void)
{
  cout << "testing grap2.txt" << endl;
  Graph g2("grap2.txt");
  //g2.dfs();
  cout << boolalpha << g2.cycle() << endl;
}



void test3 (void)
{
  cout << "testing grap3.txt" << endl;
  Graph g3("grap3.txt");
//  g3.dfs();
  cout << boolalpha << g3.cycle() << endl;
}

void test4 (void)
{
  cout << "testing grap4.txt" << endl;
  Graph g4("grap4.txt");
  //g4.dfs();
  cout << boolalpha << g4.cycle() << endl;
}

int main (void)
{
  test1();
  test2();
  test3();
  test4();
}
```

```
6
0 2 0 4 0 0
2 0 1 7 9 5
0 1 0 0 0 0
4 7 0 0 0 0
0 9 0 0 0 0
0 5 0 0 0 0
```

```
4
0 5 1 0
3 0 2 2
4 1 0 1
3 3 3 0
```

```
4
0 1 0 0
0 0 0 1
0 0 0 0
0 0 1 0
```

```
5
0 0 0 0 1
1 0 1 0 0
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
```

```
// pq.h
// This MinPriorityQueue template class assumes that the class KeyType has
// overloaded the < operator and the << stream operator.

#ifndef PQ_H
#define PQ_H

#include <iostream>
#include "heap.h"

template <class KeyType>
class MinPriorityQueue : public MinHeap<KeyType>
{
  public:
    MinPriorityQueue();              // default constructor
    MinPriorityQueue(int n);         // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                    // return the minimum element
    KeyType* extractMin();                       // delete the minimum element and return it
    void decreaseKey(int index, KeyType* key);  // decrease the value of an element
    void insert(KeyType* key);                   // insert a new element
    bool empty() const;                          // return whether the MPQ is empty
    int length() const;                          // return the number of keys
    std::string toString() const;                // return a string representation of the MPQ
    //int findIndex (KeyType * k);

    // Specify that MPQ will be referring to the following members of MinHeap<KeyType>.

    using MinHeap<KeyType>::A;
    using MinHeap<KeyType>::heapSize;
    using MinHeap<KeyType>::capacity;
    using MinHeap<KeyType>::parent;
    using MinHeap<KeyType>::swap;
    using MinHeap<KeyType>::heapify;

    /* The using statements are necessary to resolve ambiguity because
       these members do not refer to KeyType.  Alternatively, you could
       use this->heapify(0) or MinHeap<KeyType>::heapify(0).
    */
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq);

class FullError { };     // MinPriorityQueue full exception
class EmptyError { };    // MinPriorityQueue empty exception
class KeyError { };      // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

```
#include <sstream>
#include <string>
#include <iostream>
#include <exception>


/*
* notes
* minimum method to return the minimum element, but the function header makes it return a poin
ter. it works but it’s returning the address and not the int value of the smallest element
*/

/*--------------------------
* default constructor
* uses minHeap default constructor
--------------------------*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue():MinHeap<KeyType>()
{
}

/*--------------------------
* construct an empty MinPriorityQueue with capacity n
* uses minHeap constructor to construct an empty heap with capacity n
--------------------------*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n):MinHeap<KeyType>(n)
{
}
/*--------------------------
* copy constructor
* uses minHeap copy constructor
--------------------------*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType>& pq):MinHeap<KeyTy
pe>(pq)
{

}

//=============================================================================================
=======
/*--------------------------
* minimum()
* return smallest element in MinPriorityQueue
* parameters: void
* return value: KeyType*
* precondition: MPQ is not empty (exception to handle this case)
* postcondition: MPQ is unchanged
--------------------------*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum( void ) const               //note: fix exceptions
{
        if (empty())
        {
                throw EmptyError();
        }
        else
        {
                return A[0];
        }
}
```

```
/*-------------------------
* extractMin()
* delete the minimum element of MinPriorityQueue and return it
* parameters: void
* return value: KeyType*
* precondition: non-empty MPQ (exception to handle this case)
* postcondition: MPQ no longer contains smallest value
-------------------------*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin( void )
{
	//cout << "EXTRACT MIN!!!!!!!!!!" << endl;
	if (empty())
	{
		throw EmptyError();
	}
	else
	{
		KeyType* temp = A[0];
		//cout << "temp 1: " <<*temp << endl;
		A[0] = A[heapSize - 1];
		//cout << "temp 2: " << *temp << endl;
		heapSize -= 1;
		//cout << "temp 3: " << *temp << endl;
		heapify(0);
		//cout << "temp 4: " << *temp << endl;
		//cout << "END EXTRACT MIN!!!!!!!!!!" << endl;
		return temp;
	}
}

/*-------------------------
* decreaseKey()
* decrease the value of an element
* paramters: int index of value to decrease priority of, KeyType* value to decrease
* return value: void
* precondition: non-empty MPQ (exception)
* postcondition: given value has decreased in priority and its position has changed
-------------------------*/
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey( int index, KeyType* key )
{
	if (empty())
	{
		throw EmptyError();
	}
	else
	{
		A[index] = key;
		while (index > 0 and *(A[parent(index)]) > *(A[index]))
		{
			swap(index, parent(index));
			index = parent(index);
		}

	}
}

/*-------------------------
* insert
* insert a new element
* parameters: KeyType* key to insert
* return value: void
```

```
* precondition: MPQ exists
* postcondition: MPQ now contains value passed to insert
--------------------------*/
template <class KeyType>
void MinPriorityQueue<KeyType>::insert( KeyType* key )
{
        if (heapSize + 1 == capacity - 1)
        {
                throw FullError();
        }
        else
        {
                A[heapSize] = key;
                heapSize++;
                decreaseKey(heapSize-1, key);
        }
}

/*--------------------------
* empty()
* return whether the MinPriorityQueue is empty
* parameters: void
* return value: bool
* precondition: MPQ exists
* postcondition: MPQ is unchanged
--------------------------*/
template <class KeyType>
bool MinPriorityQueue<KeyType>::empty( void ) const
{
        return heapSize == 0;
}

/*--------------------------
* length
* returns length of MinPriorityQueue
* parameters: void
* return value: int
* precondition: MPQ exists
* postcondition: unchanged MPQ
--------------------------*/
template <class KeyType>
int MinPriorityQueue<KeyType>::length( void ) const
{
        return heapSize;
}


/*--------------------------
* toString
* inserts MPQ contents into string
* parameters: none
* return value: string
--------------------------*/
template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const
{
        std::stringstream ss;

        if (heapSize == 0)
        {
                ss << "[ ]";
        }
        else
```

```
        {
                ss << "[";
                for (int index = 0; index < heapSize - 1; index++)
                        ss << *(A[index]) << ", ";
                ss << *(A[heapSize - 1]) << "]";
        }
        return ss.str();
}


/*-------------------------
* overloaded cout operator
--------------------------*/

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq)
{
        stream << pq.toString();
        return stream;
}


/*template <class KeyType>
int MinPriorityQueue<KeyType>::findIndex(KeyType * k)
{
        if (empty())
                throw EmptyError();

        else
        {
                        for (int i = 0; i < length(); i++)
                        {
                                if (A[i] == *k)
                                        return i;
                        }
                        return -1;
        }
}
*/
```

```
#include <string>
#include <fstream>

const int DEFAULT_SIZE = 100;
using namespace std;
template <class KeyType>
class MinHeap
{
public:
      MinHeap(int n = DEFAULT_SIZE);          // default constructor
      MinHeap(KeyType* initA[], int n);        // construct heap from array
      MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
      ~MinHeap();                             // destructor
      void heapSort(KeyType* sorted[]);  // heapsort, return result in sorted
      std::string toString() const;     // return string representation
      MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);  // assignment operator

protected:
      KeyType **A;     // array containing the heap
      int heapSize;   // size of the heap
      int capacity;   // size of A
      void buildHeap();                 // build heap
      int leftChild(int index) { return 2 * index + 1; }  // return index of left child
      int rightChild(int index) { return 2 * index + 2; } // return index of right child
      int parent(int index) { return (index - 1) / 2; }   // return index of parent
      void swap(int index1, int index2);        // swap elements in A
      void copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
      void heapify(int index);          // heapify subheap rooted at index

      void destroy();                           // deallocate heap
};
#include "heap.cpp"
```

```
#include <iostream>
#include <sstream>


using namespace std;

//==================================================
// Default Constructor for MinHeap
// Pre-conditions:
//              none
// post-conditions:
//              It’s a heap with nothing in it,
//              capacity n, heapSize 0
// Notes:
//              Now go out into the world. Don’t return
//              until you have done all that is required.
//==================================================
template<class KeyType>
         MinHeap<KeyType>::MinHeap(int n)
{
        A = new KeyType*[n];
        capacity = n;
        heapSize = 0;
}

//==================================================
// Array Initilazation Constructor for MinHeap
// Pre-Conditions:
//              n > 0
// Post-Conditions:
//              Congratulations! It’s a MinHeap,
//              capacity n, heapSize n
// Notes:
//              Now go out into the world. Don’t return
//              until you have done all that is required.
//==================================================
template<class KeyType>
         MinHeap<KeyType>::MinHeap(KeyType* initA[], int n)
{
        capacity = n;
        heapSize = n;
        A = new KeyType*[n];
        for(int i = 0; i < n; i++)
        {
                A[i] = initA[i];
        }
        buildHeap();
}

//==================================================
// Copy Constructor for MinHeap
// Pre-Conditions:
//              none
// Post-Conditions:
//              Congratulations! It’s a MinHeap,
//              capacity heap.capacity,
//              heapSize heap.heapSize
// Notes:
//              Now go out into the world. Don’t return
//              until you have done all that is required.
//==================================================
template<class KeyType>
         MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap)
```

```
{
        copy(heap);
}

//=================================================
// Destructor for MinHeap
// Pre-Conditions:
//              None
// Post-Conditions:
//              None
// Notes:
//              And when you do return, I am here. And I
//              will destroy you. It is all I know.
//=================================================
template<class KeyType>
         MinHeap<KeyType>::~MinHeap()
{
        destroy();
}

//=================================================
// Sorting Algorithm: heapSort
// Pre-Conditions:
//              The heap must be a MinHeap
// Post-Conditions:
//              sorted is now sorted in ascending order
//=================================================
template<class KeyType>
void MinHeap<KeyType>::heapSort(KeyType* sorted[])
{
        sorted = new KeyType*[capacity];
        //buildHeap();
        for(int i = capacity - 1; i >= 0; i--)
        {
                sorted[i] = A[0];
                swap(0,i);
                heapSize--;
                heapify(0);
        }
        heapSize = capacity;
}

//=================================================
// Assignment operator
// Pre-Conditions:
//              none
// Post-Conditions:
//              returns a new heap just like the heap
//              which was passed in
//=================================================
template<class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap)
{
        destroy();
        copy(heap);
        return *this;
}

//=================================================
// String converter
// Pre-Conditions:
//              none
// Post-Conditions:
```

```
//              returns a string of the array in which
//              the heap is stored
//====================================================
template<class KeyType>
string  MinHeap<KeyType>::toString() const
{
        stringstream stm;
        stm << "{";
        for(int i = 0; i < heapSize - 1; i++)
        {
                stm << *(A[i]) << ", ";
        }
        if(heapSize != 0)
                stm << *(A[heapSize - 1]) << "}";
        else
                stm << "}";

        return stm.str();
}

//====================================================
// Makes a heap into a min heap
// Pre-Conditions:
//              Both children must be roots of a Min-Heap
// Post-Conditions:
//              The heap is a Min-Heap (if the
//              Pre-Condition is satisfied)
//====================================================
template<class KeyType>
void  MinHeap<KeyType>::heapify(int index)
{
        int l = leftChild(index);
        int r = rightChild(index);
        int min;
        if(l < heapSize && *(A[index]) > *(A[l]))
                min = l;
        else
                min = index;
        if(r < heapSize && *(A[min]) > *(A[r]))
                min = r;
        if(min != index)
        {
                swap(index, min);
                heapify(min);
        }
}

//====================================================
// Builds a MinHeap
// Pre-Conditions:
//              none
// Post-Conditions:
//              the heap is definitely a Min-Heap
//====================================================
template<class KeyType>
void  MinHeap<KeyType>::buildHeap()
{
        heapSize = capacity;
        for(int i = heapSize / 2 - 1; i >= 0; i--)
        {
                heapify(i);
        }
}
```

```
//==================================================
// Swaps two items
// Pre-Conditions:
//              The indices are valid
// Post-Conditions:
//              The values at the indices
//              have been swapped
//==================================================
template<class KeyType>
void  MinHeap<KeyType>::swap(int index1, int index2)
{
        KeyType* temp = A[index1];
        A[index1] = A[index2];
        A[index2] = temp;
}

//==================================================
// copies one heap into another
// Pre-Conditions:
//              none
// Post-Conditions:
//              This heap is just like the one passed in.
//              capacity heap.capacity,
//              heapSize heap.heapSize
//==================================================
template<class KeyType>
void  MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap)
{
        A = new KeyType*[heap.capacity];
        for(int i = 0; i < heap.capacity; i++)
                A[i] = heap.A[i];
        capacity = heap.capacity;
        heapSize = heap.heapSize;
}

//==================================================
// The End
// Pre-Conditions:
//              none
// Post-Conditions:
//              none
// Notes:
//              But don't feel betrayed or singled out.
//              I heed nothing. I leave nothing. All are
//              destroyed. Everything shall be deleted.
//==================================================
template<class KeyType>
void  MinHeap<KeyType>::destroy()
{
        delete A;
}
```

```
// list.h
// Jessen Havill

#ifndef LIST_H
#define LIST_H

#include <cstdlib>
#include <iostream>

template <class T>
class Node
{
 public:

   T *item;
   Node<T> *next;

   Node();
   Node(T *initItem);
   Node(T *initItem, Node<T> *initNext);
};

template <class T> class List;

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& list);

template <class T>
class List
{
 public:

   List();                                // default constructor
   List(const List<T>& src);              // copy constructor
   ~List();                               // destructor

   void append(T *item);                  // append a new item to the end of the list
   int length() const;                    // return the number of items in the list
   int index(const T& item) const;        // return index of value item, or -1 if not found
   T *get(const T& item) const;           // return pointer to the item equal to parameter item
   void insert(int index, T *item);       // insert item in position index
   T *pop(int index);                     // delete the item in position index and return it
   void remove(const T& item);            // remove the first occurrence of the value item

   T *operator[](int index) const;              // indexing operator
   List<T>& operator=(const List<T>& src);      // assignment operator
   List<T>& operator+=(const List<T>& src);     // concatenation operator

   std::string toString() const;

 private:

   Node<T> *head;                         // head of the linked list
   int count;                             // number of items in the list

   void deepCopy(const List<T>& src);
   void deallocate();                     // deallocate the list
   Node<T>* _find(int index) const;       // return a pointer to the node in position index

   friend std::ostream& operator<< <T>(std::ostream& os, const List<T>& list);
};

class IndexError { };
```

```
class ValueError { };

#include "list.cpp"

#endif
```

```
// list.cpp
// Jessen Havill

#include <sstream>

template <class T>
Node<T>::Node()
{
   item = NULL;
   next = NULL;
}

template <class T>
Node<T>::Node(T *initItem)
{
   item = initItem;
   next = NULL;
}

template <class T>
Node<T>::Node(T *initItem, Node<T> *initNext)
{
   item = initItem;
   next = initNext;
}

template <class T>
List<T>::List()
{
        head = NULL;
        count = 0;
}

template <class T>
List<T>::List(const List<T>& src)
{
        deepCopy(src);
}

template <class T>
List<T>::~List()
{
        deallocate();
}

template <class T>
List<T>& List<T>::operator=(const List<T>& src)
{
        deallocate();
        deepCopy(src);

        return *this;
}

template <class T>
int List<T>::length() const
{
        return count;
}

template <class T>
int List<T>::index(const T& item) const
{
```

```
        int index = 0;
        Node<T> *node = head;
        while ((node != NULL) && (!(*(node->item) == item)))
        {
                node = node->next;
                index++;
        }

        if (node == NULL)
                return -1;
        else
                return index;
}

template <class T>
T *List<T>::get(const T& item) const
{
        Node<T> *node = head;
        while ((node != NULL) && (!(*(node->item) == item)))
                node = node->next;

        if (node == NULL)
                return NULL;
        else
                return node->item;
}

template <class T>
void List<T>::append(T *item)
{
        Node<T> *node,
                *newNode;

        newNode = new Node<T>(item);

        if (head != NULL)
        {
                node = _find(count - 1);
                node->next = newNode;
        }
        else
                head = newNode;

        count++;
}

template <class T>
void List<T>::insert(int index, T *item)
{
        if ((index < 0) || (index > count))
                throw IndexError();

        Node<T> *node;

        if (index == 0)
                head = new Node<T>(item, head);
        else
        {
                node = _find(index - 1);
                node->next = new Node<T>(item, node->next);
        }
        count++;
}
```

```
template <class T>
T *List<T>::pop(int index)
{
        if ((index < -1) || (index >= count))
                throw IndexError();

        if (index == -1)
                index = count - 1;

        Node<T> *node, *dnode;
        T *item;

        if (index == 0)
        {
                dnode = head;
                head = head->next;
                item = dnode->item;
                delete dnode;
        }
        else
        {
                node = _find(index - 1);
                if (node != NULL)
                {
                        dnode = node->next;
                        node->next = node->next->next;
                        item = dnode->item;
                        delete dnode;
                }
        }
        count --;
        return item;
}

template <class T>
T* List<T>::operator[](int index) const
{
        if ((index < 0) || (index >= count))
                throw IndexError();

        Node<T> *node = _find(index);
        return node->item;
}

template <class T>
void List<T>::deepCopy(const List<T>& src)
{
        Node<T> *snode, *node;

        snode = src.head;
        if (snode != NULL)
        {
                node = head = new Node<T>(snode->item);
                snode = snode->next;
        }
        while (snode != NULL)
        {
                node->next = new Node<T>(snode->item);
                node = node->next;
                snode = snode->next;
        }
```

```
        count = src.count;
}

template <class T>
void List<T>::deallocate()
{
        Node<T> *node, *dnode;

        node = head;
        while (node != NULL)
        {
                dnode = node;
                node = node->next;
                delete dnode;
        }
}

template <class T>
void List<T>::remove(const T& item)
{
        if (head == NULL)
                return;

        Node<T> *toDelete;

        if (*(head->item) == item)
        {
                toDelete = head;
                head = head->next;
                delete toDelete;
                count--;
        }
        else
        {
          Node<T> *node = head;
          while ((node->next != NULL) && (!(*(node->next->item) == item)))
                  node = node->next;

          if (node->next != NULL)
          {
                  toDelete = node->next;
                  node->next = node->next->next;
                  delete toDelete;
                  count--;
          }
        }
}

template <class T>
Node<T>* List<T>::_find(int index) const  // used by append, insert, [], pop
{
        if ((index < 0) || (index >= count))
                throw IndexError();

        Node<T> *node = head;
        for (int i = 0; i < index; i++)
                node = node->next;

        return node;
}

template <class T>
std::string List<T>::toString() const
```

```
{
        std::stringstream ss;
        Node<T> *node = head;

        while (node != NULL)
        {
                ss << *(node->item) << " ";
                node = node->next;
        }
        return ss.str();
}

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& list)
{
        os << list.toString();

        return os;
}
```